Chapter 7

Data Types and Declarations

The values of both constants and variables have data types. Data types specify the amount of storage required and how to interpret the data object in that storage space. This chapter discusses the following topics in respect to to data types:

- Constants
- Variables
- Integers
- Floating-point values
- Pointers
- Enumerated types
- Arrays
- Characters
- Structures and unions
- The **void** keyword
- The **typedef** keyword
- Interpreting variable declarations

7.1 Constants

You can represent data in VAX C using constants. A constant is a primary expression with a defined value that does not change. You may represent a constant in a literal form, which contains the explicit numbers, letters, and operators that comprise the constant. You may define a symbol to represent the constant value. (For more information about symbolic representation of constants, see Chapter 9.) Constants, like all data in VAX C, have data types. The data type determines the amount of storage needed and determines how to interpret the stored object or constant value. The compiler determines the data type of constants by the way their values are represented in the source code.

7.2 Variables

You can also represent data in VAX C using variables, whose values can change throughout the execution of the program. You must declare all variables used in a program. When you declare a variable, you specify the data type of the stored object. In VAX C, an object is a value requiring storage.

Declarations determine the size of a storage allocation; definitions initiate the allocation of storage. You can declare and define variables. Most variable declarations are also definitions because storage is allocated at that point in the program. To declare a variable, specify the data type. To define a variable, assign the variable the proper storage class and place the variable declaration within the program structure. Also, if you can initialize a variable in the declaration, the variable is defined. For more information about variable definitions, scope, and storage allocation, see Chapter 8.

There are two kinds of variables: scalar and aggregate variables. Scalar variables have objects that you can manipulate arithmetically in their entirety. These objects are single characters, individual numbers, and pointers. Aggregate variables are data structures (arrays, structures, and unions) that are comprised of distinct elements (members) that you can declare to be either a scalar or aggregate data type.

7.2.1 Data-Type Keywords

To declare or define variables, you need to know the VAX C keywords associated with each data type. Table 7–1 lists the VAX C data-type keywords by classification.

Table 7–1: VAX C Data-Type Keywords

Scalar Keywords	Aggregate Keywords	Other Type Keywords
int	**struct**	**void**
long	**union**	
unsigned	**variant_struct**	
short	**variant_union**	
char		
float		
double		
num		

In the following sections, the keywords and operators used to declare variables of given data types are listed in the section header for easy reference.

VAX C also supports the **const** and **volatile** type modifiers. For more information about these type modifiers, see Chapter 8.

7.2.2 Format of a Variable Declaration

A variable declaration can be composed of the following items:

- Data-type specifiers such as a data type or data-type modifier keyword, one structure, union, or **enum** tag, and if necessary, a **typedef** name.

  Any of these give the data type of the declared object.

- An optional storage-class keyword.

A storage-class keyword affects the scope of a variable and determines how it is stored. If you omit the storage-class keyword, there is a default storage class that depends on the physical location of the declaration in the program. The positions of the storage-class keywords and the data-type keywords are interchangeable.

- Declarators, which list the identifiers of the declared objects and may contain operators that declare a pointer, function, or an array of objects of the declared type.
- Initializers for each declared object or aggregate element giving the initial value of a scalar variable or the initial values of structure members or array elements.

  An initializer consists of an equal sign (=) followed by a single expression or a comma-list of one or more expressions in braces.

Consider the following example:

```
int  var_number = 10;
```

The declaration both declares and defines the integer variable, var_number, which has an initial value of 10. The **int** keyword specifies the amount of storage needed on a VAX system for an integer. The identifier var_number follows. The equality operator (=) initializes the variable with the literal constant 10; for the initialization to take place, storage is allocated and the variable is defined. Declarations must end in a semicolon (;).

The variable declaration in the previous example was not difficult to interpret, but even experienced VAX C programmers have difficulty interpreting complex variable declarations. See Section 7.15 for more information about interpreting the VAX C variable declarations.

7.3 Integers (int, long, short, char, and unsigned)

You can declare integer variables with the **int**, **long**, **short**, **char**, and **unsigned** keywords. The following is an example of an integer declaration:

```
int x;
```

Character variables are declared with the **char** keyword. An example of a character declaration with the initialization of a character variable is as follows:

```
char ch = 'a';
```

Table 7–2 specifies the sizes and ranges of integers.

Table 7–2: The Size and Range of VAX C Integers

Keyword	Size	Range
int, **long**, and **long int**	32 bits	–2,147,483,648 to 2,147,483,647
unsigned and **unsigned int**	32 bits	0 to 4,294,967,295

(continued on next page)

Table 7–2 (Cont.): The Size and Range of VAX C Integers

Keyword	Size	Range
short and **short int**	16 bits	–32,768 to 32,767
unsigned short	16 bits	0 to 65,535
char	8 bits	–128 to 127
unsigned char	8 bits	0 to 255

The following sections describe the constants that you can assign to the integer variables.

7.3.1 Integer Constants

There are three types of integer constants; decimal, hexadecimal, and octal. Integer constants can consist of the characters 0 to 9, a to f (for hexadecimal integers), and A to F (also for hexadecimal integers).

Integer constants can also include an optional suffix consisting of the characters x, X, l, L, u, or U. Characters x and X specify hexadecimal integers. Characters l and L specify **long** integers (of 4 bytes or 1 longword). Characters u and U specify **unsigned** integers. Characters l or L and u or U can be combined to specify an **unsigned long** integer.

On some other implementations of the C language, values of the **int** data type require 16 bits of storage. On VAX architecture, values of the **int** data type require 32 bits of storage, the same amount of storage as values of the **long** data type. VAX C supports the L suffix only for the sake of program portability.

You can specify integer constants in decimal, octal, and hexadecimal radixes. An integer constant is assumed to be decimal unless it begins with 0 or 0x; if it begins with 0, it is assumed to be octal; if it begins with 0x, it is assumed to be hexadecimal.

In octal constants, the digits 8 and 9 have the octal values 010 and 011, respectively. For instance, the octal number 039 is equal to $3 * 8 + 9$, or decimal value 33; the octal number 080 is equal to $8 * 8 + 0$, or, decimal value 64.

Even though VAX C supports the digits 8 and 9 in octal constants, avoid using these octal constants to be compatible with other implementations of the C language.

Integer constants must not include a decimal point; constants with a decimal point are of type **double**. Integer constants that exceed a longword are treated as programming errors.

Character constants such as `'a'` and `'$'` are also valid integer constants. Their integer values in VAX C are the values of the corresponding ASCII codes.

Some examples of valid integer constants are as follows:

```
133L                    /* Long decimal integer                 */
0x17A                   /* Hexadecimal integer                  */
056                     /* Octal integer                        */
'a'                     /* Decimal 97                           */
'$'                     /* Decimal 36                           */
```

The following examples show invalid integer constants:

```
143.                    /* Includes a decimal point              */
3333333333              /* Out of range for int                  */
+33333                  /* '+' is an invalid character           */
77af                    /* Hexadecimal constants must be         *
                         *  prefixed with "0x"                   */
```

7.3.2 Character Constants

A character constant is a value, requiring at least 8 bits (1 byte) or at most 32 bits (1 longword) of memory, that is enclosed in apostrophes. Character constants can be a single ASCII character, as in the following example:

```
char ch = 'a';          /* Lowercase letter 'a' is a constant    *
                         *  assigned to ch.                      */
```

The character constant 'a' has the ASCII value 97. If the value of a character constant is not large enough to fill 32 bits of memory, the compiler stores the character or characters in the low-order byte(s) and pads the remaining bytes with NUL characters ('\0').

Character constants do not have to be single characters, as shown in the following example (please note that this is VAX C specific, and not portable):

```
int l_word = 'a:cd' /*  This constant contains 4 characters    */

printf("%c\n", l_word);
printf("%.4s", &l_word); /*  String with maximum 4 characters  */
```

Sample output from the previous example is as follows:

```
% example RET
a
a:cd
%
```

If you print variable l_word as a character, the **printf** function prints only the character located in the low-order byte of the integer allocation. To print all of the characters in the longword allocated to the variable, you have to print the variable as a string and pass the address of the integer variable as an argument. If you print the integer variable as a string, be sure to specify a precision of at most 4, since you can never be sure if the next byte in the string is a terminating NUL character.

The apostrophe (') and quotation mark (") are significantly different punctuation marks in VAX C, indicating a character constant and a string constant, respectively. One context in which the difference is important is in an argument list. If you specify a function argument as a string, and want to pass a character constant, you must enclose the character in quotation marks, not apostrophes, even if the string is only one to four characters in length. See Section 7.11 for more information about character-string constants.

7.3.3 Escape Sequences

In VAX C, escape sequences are character strings that represent a single printing or nonprinting character. The term escape sequence does not designate a string beginning with the ASCII character ESC, as in VT100 escape sequences. Table 7–3 presents the escape sequences that specify the nonprinting characters, the apostrophe, and the backslash (\).

Table 7–3: VAX C Escape Sequences

Character	Mnemonic	Escape Sequence
newline	NL	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
apostrophe	'	\'
quotes	"	\"
bit pattern	ddd	\ddd or \xddd

An escape sequence, such as \n, denotes a single character.

The form \ddd specifies any byte value (usually an ASCII code), where the digits ddd are one to three octal digits. The octal digits are limited to 0 through 7. A common use is to specify the ASCII NUL character, as follows:

```
'\0'
```

Similarly, the form \xddd specifies any byte value (usually an ASCII code), where the digits ddd specify one to three hexadecimal digits.

The following are examples of valid escape sequences of the form \ddd and \xddd. Both of these escape sequences are used to specify an a-umlaut (ä) in octal and hexadecimal digits, respectively.

```
'\344'
'\xe4'
```

If the character following the backslash in an escape sequence is illegal, the backslash is ignored; that is, the character constant's value is the same as if the backslash were not present.

7.4 Floating-Point Numbers (float and double)

When declaring floating-point variables, you determine the amount of precision needed for the stored object. In VAX C, you can have either single-precision or double-precision variables. If you choose single precision, you use the F-floating format. If you choose double precision, you have the choice of using either the D_floating or G_floating formats.

Table 7–4 specifies the sizes and ranges of real numbers.

Table 7–4: The Size and Range of C Floating-Point Numbers

Keyword	Size	Range	Precision
float	32 bits	$0.29 * 10^{-38}$ to $1.7 * 10^{38}$	7 decimal digits

(continued on next page)

Table 7–4 (Cont.): The Size and Range of C Floating-Point Numbers

Keyword	Size	Range	Precision
double D_Floating	64 bits	$0.29 * 10^{-38}$ to $1.7 * 10^{38}$	16 decimal digits
double G_Floating	64 bits	$0.56 * 10^{-308}$ to $0.899 * 10^{308}$	15 decimal digits

Use the **float** keyword to declare a single-precision, floating-point variable, represented internally in the VAX F_floating point binary format.

The **double** keyword declares a double-precision, floating-point variable. You can use the **double** and **long float** keywords interchangeably. However, do not use **long float** to avoid conflict with other implementations of the C language. There are two representations of the VAX C data type **double**: D_floating and G_floating.

The G_floating precision, approximately 15 digits, is less than that of variables represented in D_floating format. The fractional portion of the variable may contain one more digit, but the integral portion of the variable must contain one less digit.

The default representation of the data type **double** is D_floating. The G_floating representation is chosen by compiling the program with the **–Mg** option on the compile command. (For more information about the compilation command line, see Chapter 2.) Do not link modules compiled with the D_floating representation with modules compiled with the G_floating representation.

7.5 Floating-Point Constants

A floating-point constant has an integral part (a decimal point), a fractional part (the letter e or E), and an optionally signed integer exponent. The integral and fractional parts consist of decimal digits; you may omit either the integral or fractional part. You may omit either the decimal point with the following digits or the E<exponent>, but not both.

Floating-point constants can also include an optional suffix consisting of the characters l, L, f, or F. Constants without suffixes, or with the l or L suffix, are of type **double**. Constants with the f or F suffix are of type **float**.

The following examples are floating-point constants:

```
3.0e10
3.0E-10
3.0e+10
3E10
3.0
.120e2
.120
```

7.6 Pointers (*)

Pointers in VAX C are variables that contain the 32-bit addresses of other objects. They are declared with the asterisk operator and the data type of the object that it points to, as in the following example:

```
int *px;
```

Identifier px is declared as a pointer to a variable of type **int**; the construct *px is treated as a variable of type **int**. An expression such as *px yields the integer that px points to.

Unless an **[extern]** or **static** pointer variable is initialized, it is a null pointer. A null pointer is a pointer variable that has been assigned the integer constant 0. The contents of an uninitialized **auto** pointer are undefined.

In certain arithmetic expressions, the compiler uses the size of the object of the pointer. For example, if px is a pointer to an integer, px + 1 evaluates to the next integer address, 4 bytes after px. If px is a pointer to **char**, px + 1 yields the next **char** address, 1 byte after px. The compiler uses the type of the pointed object to scale the arithmetic.

A different result occurs with an expression such as the following example:

```
*px + 1
```

This expression evaluates to the value of the object that px points to plus one.

Some contexts may require a pointer of a particular type. This is necessary, for example, when a function requires that an argument be passed by reference.

The unary asterisk (*) is also the indirection operator in VAX C. The unary asterisk operates as follows:

```
x = *px;
```

This statement assigns the value of the object pointed to by px to variable x. Since you can use the asterisk in any sort of declarator, you can have pointers to scalars, to functions, to other pointers, to structures, and so forth.

Use the ampersand (&) operator to take the address of an object. Consider the following example:

```
px = &x;
```

This statement assigns the address of variable x to pointer px. After an assignment such as this, a reference to *px yields the value of x.

Do not apply the ampersand operator to constants, to **register** variables, to function identifiers, or to array identifiers. Though the compiler will allow it, it is not portable.

The compiler stores constant values in a read-only program section (psect), so that attempts to change the value by applying the ampersand operator will result in an error. VAX C allows the application of the ampersand operator to constants so that you can pass constants, as arguments, to routines. For more information about psects, see Chapter 8.

If you do apply the ampersand to **register** variables, the optimizing section of the compiler prevents any promotion to registers.

If you apply the ampersand to function or array identifiers, VAX C issues a message, since asking for the address of an expression returning an address is redundant.

7.6.1 void Pointers

The **void** pointer is a pointer that does not have a specified data type to describe the object to which it points. In effect, this is a generic pointer. (In the past, VAX C programmers have used **char** * to define generic pointers; this practice is now discouraged for portability reasons.)

You can assign a pointer of any type to a **void** pointer without a cast (see Section 6.4.5 for more information on the cast operation). For example, you can use this type of pointer in function calls, in function arguments, or in function prototypes when the parameter or return value is a pointer of an unkown type. Consider the following example:

```
main()
{
   void  *generic_pointer;
   .
   .
   .
/* If the return value can be a pointer to many types . . .   */
   generic_pointer = func_returning_pointer( arg1, arg2, arg3 );
   .
   .
   .
}
```

The following statements are also valid:

```
main()
{
float *float_pointer;
void  *void_pointer;
   .
   .
   .
float_pointer = void_pointer;
/*  Or . . .    */
void_pointer  = float_pointer;
   .
   .
   .
```

See Section 4.8.2 for information about using **void** in function definitions.

7.7 Enumerated Types

An enumerated type is a user-defined data type that is not derived from the fundamental types. Each listed enumerator is associated with an incremented integer constant starting with 0. The following example shows the declaration of a variable and an enumeration type, or tag:

```
enum  shades
   {
      out, verydim, dim, prettybright, bright
   }  light;
```

This declaration defines the variable light to be of an enumerated type shades. The variable can assume any of the enumerated values.

The tag shades becomes the enumeration tag of the new type; out, verydim . . . bright are the enumerators with values 0 through 4. These enumerators are the constant values of the type shades and can be used wherever constants are valid.

If you have declared the tag, use the tag as a reference to that enumerated type, as in the following declaration:

```
enum  shades  light1;
```

The variable light1 is an object of the enumerated data type shades.

An **enum** tag can have the same spelling as other identifiers in the same program, including variable identifiers and member names in structures and unions, because the meanings are distinguished by context. However, **enum** constant names must have unique spellings. VAX C allows forward references to **enum** tags that have not been declared yet in the source code, but are declared further in the program.

Internally, each enumerator is associated with an integer constant; the compiler gives the first enumerator the value 0 by default, and the remaining enumerators are incremented by the value 1, as they are read from left to right. Any enumerator can be set to a specific integer constant value. The enumerators to the right of such a construct (unless they are also set to specific values) then receive values that are one greater than the previous value. Consider the following example:

```
enum spectrum
   {
      red, yellow = 4, green, blue, indigo, violet
   }  color2;
```

This declaration gives red, yellow, green, blue, . . . , the values 0, 4, 5, 6, . . .

Examining the value of a variable like color2 displays an integer, not a string such as red or yellow. They are stored internally as integers, but you should regard enumerated data types as being distinct from the fundamental types.

Type mismatches between the enumerated and fundamental types, or between different enumerated types, are errors. The following example is not valid:

```
enum
   {
      red, orange, yellow, green, blue, indigo, violet
   }  color1;

enum  illum
   {
      out, verydim, dim, prettybright, bright
   }  light;

light = red;
```

The enumerators red and light have different enumerated types.

The following example is also invalid:

```
enum  illum
   {
      out, verydim, dim, prettybright, bright
   }  light;

light = 1;
```

Value 1 is not an enumerated value for variable light.

To perform valid mixed-type operations, use the cast operator. Consider the following example:

```
/*  Both evaluate to verydim (1)  */

light = (enum illum) (out + (enum illum) red);
light = (enum illum) 1;
```

The cast operation (enum illum) causes the compiler to treat **enum** constant red and integer constant 1 as values of enumerated type illum.

Variables and enumerators of enumerated types take on various storage classifications when used with the **globaldef** and **globalref** storage-class keywords. For more information about using these storage-class keywords with enumerated types, see Chapter 8.

7.8 Arrays ([ ])

You declare arrays with the square bracket operators ([ ]), as in the following declaration of a 10-element array of integers called table_one:

```
int table_one[10];
```

The **int** type specifier gives the data type of the elements. The elements of an array can be of any scalar or aggregate data type. The identifier table_one specifies the name of the array. The constant expression gives the number of elements in a single dimension. Array subscripts in VAX C begin with the integer 0 (not 1); they must be integral. In the previous example, the first element is table_one[0] and the last element is table_one[9]. Unpredictable results can occur if you specify a subscript larger than or equal to the declared dimension bound; you would then be accessing objects outside of the memory allocated to the array. It is not recommended that you use array subscripts as shown in the following example:

```
int table_one[10];

table_one[10] = 69;
table_one[5]  = table_one[11];
```

VAX C supports multidimensional arrays, which are arrays declared as an array of arrays. Consider the following example:

```
int table_one[10][2];
```

Variable table_one is a 2-dimensional array containing 20 integers. You can use C operators to form expressions with specific elements of an array, as follows:

```
++table_one[0][0];        /* Increment first element          */
```

In VAX C, arrays are stored in row-major order. The element table_one[0][0] immediately precedes table_one[0][1], which in turn immediately precedes table_one[0][2].

When you declare an array, either single- or multidimensional, the integer constant is optional in the first pair of brackets. Omitting the constant expression is useful in the following cases:

- If the array is external, its storage is allocated by a remote definition. Therefore, you can omit the constant expression for convenience when the array name is declared, as in the following example:

```
extern int array1[];
first_function()
{
    .
    .
    .
}
```

In a separate compilation:

```
int array1[10];
second_function()
{
   .
   .
   .
}
```

For more information about external data declarations, see Chapter 8.

- If the declaration of the array includes initializers, you can omit the size of the array, as in the following example:

```
char  array_one[] = "Shemps"
char  array_two[] = { 'S', 'h', 'e', 'm', 'p', 's', '\0' };
```

The two definitions initialize variables with identical elements. These arrays have seven elements: six characters and the null character (\0), which terminates all character strings. VAX C determines the size of the array from the number of characters in the initializing character-string constant or initialization list.

- If you use the array as a function parameter, it must be defined in the calling function. However, the declaration of the parameter in the called function can omit the constant expression within the brackets. The address of the beginning of the array is passed and subscripted references in the called function can modify elements of the array.

The following example shows how to use an array in this manner:

```
main()
{
                              /* Initialize array          */
   static char arg_str[] = "Thomas";
   int    sum;
   sum = adder(arg_str); /* Pass address of array      */
      .
      .
      .
}

/*  Function adds ASCII values of letters in array     */

adder(param_string)
char param_string[];
{
    int i, sum = 0;          /* Incrementer and sum       */
                             /* Loop until NUL char       */
    for (i = 0; param_string[i] != '\0'; i++)
       sum += param_string[i];
    return sum;
}
```

After the function adder is called, parameter param_string receives the address of the first character of argument arg_str, which can then be manipulated in adder. The declaration of param_string serves only to give the type of the parameter, not to reserve storage for it.

7.9 Initializing Arrays

When initializing array elements, separate the values with a comma and delimit the comma list with braces ({}). The rules for specifying a comma-list are as follows:

- If the initializer for an array begins with a left brace ({), then the following comma-list provides initial values for the array elements. The list of initializers can end with a comma, which is ignored. The number of initializers cannot be greater than the number of elements.
- If the initializer does not begin with a left brace, then only enough elements are taken from the initializer list to supply values to the array's elements. In this case, there can be more initializers than there are elements, and any remaining values in the list are left to initialize the next aggregate.

Initialize a single-dimension array as follows:

```
static int ex_array[5] = { 1, 22, 333, 4444, 55555 };
```

Initialize a multidimensional array as follows:

```
static int ex_array[2][5] =
   {
      { 1, 22, 333, 4444, 55555 },
      { 5, 4, 3, 2, 1 }
   };
```

The element ex_array[0][0] has a value of 1, ex_array[0][1] has a value of 22, . . . , ex_array[1][0] has a value of 5, ex_array[1][1] has a value of 4, . . . , and so forth.

Another way to initialize the same array is as follows:

```
int ex_array[2][5] = { 1, 22, 333, 4444, 55555, 5, 4, 3, 2, 1 };
```

VAX C initializes the elements in row-major order. The leftmost brace determines the row number of a multidimensional array. Elements in row 0 are initialized before elements in row 1.

You can omit elements in an initialization as follows:

```
static int ex_array[2][5] =
   {
      { 1, 22, 333, 4444 }
   };
```

The element ex_array[0][0] has the value 1, ex_array[0][1] has the value 22, ex_array[0][2] has the value 333, and ex_array[0][3] has the value 4444. The last element in row 0, since ex_array was declared to have a storage class of **static**, is initialized with 0. All of the elements in the second row, which are not specified in the initialization, are initialized with 0. For more information about the static storage class, see Chapter 8.

NOTE

You cannot initialize array elements without initializing all preceding elements. The following initialization is not valid:

```
example[3] = { 1 , , 3 };
```

You must initialize the first and second elements before initializing the third.

7.10 Character-String Variables (char * and char [ ])

VAX C treats character strings as arrays; they are treated as the address in memory of the first character in the string. There are several ways to declare character-string variables. You can declare a character string by designating a pointer to the first character of that string, as in the following example:

```
char  *ex_string  =  "thomasina";
```

This expression copies an address, not a string, to variable ex_string. The object to which ex_string points, a character-string constant, ends with the NUL character (\0).

You can declare character-string variables as you would declare an array. For example:

```
char string_one[] = "thomasina";
char string_2[10] = "thomasina";
```

See Section 7.9 for more information about declaring and initializing character-string variables.

To copy one string to another, you must use the **strcpy** or the **strncpy** functions, as follows:

```
main()
{
   #include <stdio.h>
   char ex_string[26];
                                        /* Copy string into array   */
   strcpy(ex_string, "Character-string constant");
   printf("%s\n", ex_string);
      .
      .
      .
}
```

7.11 Character-String Constants

A character-string constant is a series of characters enclosed in quotation marks (" "). Consider the following example:

```
"This is a string constant *** "
```

It has the data type of an array of **char**. The string is initialized with the given characters. The compiler terminates the string with a NUL character (\0). There is no formal limit to the length of a string constant. The actual limit to a string constant's length in VAX C is 65,535 characters. All strings, even when written identically, are distinct objects.

The apostrophe (') and quotation mark (") are significantly different punctuation marks in VAX C. See Section 7.3.2 for more information.

The following rules apply to the characters used in character-string constants:

- You can use all characters, including the escape sequences, in strings.
- You must precede a quotation mark within a string with a backslash (\).
- A backslash followed immediately by a newline is ignored, allowing long strings to be continued in the first column of the next line.

- You can use character strings to initialize variables of storage class **auto** as well as variables of other storage classes.

7.12 Structures and Unions (struct and union)

Structures and unions share the following characteristics:

- Their members can be variables of any type, including other structures and unions or arrays. A member can also consist of a specified number of bits, called a field.
- The only valid operators with structures and unions are the simple assignment (=) and **sizeof** operators. In particular, structures and unions may not appear as operands of the equality (==), inequality (!=), or cast operator.
- They can be assigned to other structures and unions with the assignment operator (=). The two structures or unions in the assignment must have the same length.
- They can be passed to and returned by functions. The argument must have the same length as the function parameter. A structure or union is passed by value, just like a scalar variable; that is, the entire structure or union is copied into the corresponding parameter.

NOTE

When you pass structures as arguments, they may or may not terminate on a longword boundary. If they do not, VAX C aligns the following argument on the next longword boundary.

The difference between structures and unions lies in the way their members are stored and initialized as follows:

- The members of a structure all begin at different offsets from the base of the structure. The offset of a particular member corresponds to the order of its declaration; the first member is at offset 0. Each successive nonfield member of a structure begins at the next byte boundary that matches the alignment appropriate to its type. For example, a short integer is aligned on a 2-byte boundary and a long integer is aligned on a 4-byte boundary. Gaps can appear in a structure as the compiler tries to achieve this alignment.

  Structures also observe the following restriction: the length of a structure must be a multiple of the greatest alignment requirement of any of its members. Thus, a structure that contains characters, short integers, and longwords will be a multiple of four in length to match the multiple of four bytes for the longword.
- In a union, every member begins at offset 0 from the address of the union. The size of the union in memory is the size of its largest member. When the single storage space allocated to the union contains a smaller member, the extra space between the end of the smaller member and the end of the allocated memory remains unaltered. The rules for alignment of union members is the same as for structure members.
- Structures can be initialized; unions cannot.

The VAX C compiler aligns structure members on natural type boundaries by default. You can turn off member alignment with the **#pragma nomember_alignment** directive.

7.12.1 Declaring a Structure or Union

To declare structures and unions, use the **struct** or **union** keywords. You can follow the **struct** or **union** keywords with a tag, which gives a name to the structure or union type in much the same way that an **enum** tag gives a name to the enumerated type. You can then use the tag with the **struct** or **union** keywords to declare variables of that type without specifying individual member declarations again.

Two structures (or two unions) cannot have the same tag, but the tags can be the same as the identifiers used for variables and function names. Tags can also have the same spellings as member names. The compiler distinguishes them by context. The scope of a tag is the same as the scope of the declaration in which it appears.

The tag is followed by braces ({}) that enclose a list of member declarations. Each declaration in the list gives the data type and name of one or more members. The names of structure or union members can be the same as other variables, function names, or members in other structures or unions. The compiler distinguishes them by context. In addition, the scope of the member name is the same as the scope of the declaration in which it appears.

You can place declarations after the list of member declarations, which name and reserve storage for structure or union objects.

Structure or union declarations can take one of five forms, as follows:

- If a declaration includes only a tag and a list of member declarations, then the list of member declarations defines the tag to be a data type by which other objects can be declared. For example:

```
struct person
   {
      char first[20];
      char middle[3];
      char last[30];
   };
```

- When a declaration includes a tag, a list of member declarations, and a list of identifiers, the identifiers become objects of the structure type and the tag is considered to be a shorthand notation, or mnemonic, for the structure type. Consider the following example:

```
struct person
   {
      char first[20];
      char middle[3];
      char last[30];
   }  george, mary ;
```

- If the tag is omitted, the structure or union definition applies only to the variable identifiers that follow in the declaration. Consider the following example:

```
struct
   {
      char first[20];
      char middle[3];
      char last[30];
   }  george, mary;
```

- The fourth form uses the tag to see a structure or union defined in another declaration. The definition is then applied to the variable identifiers that follow the tag name in the declaration. Consider the following example:

```
struct   person   george,mary;
```

- The fifth form uses only the **struct** or **union** keyword and the tag to override other identical tags in scope, and to reserve the tag for a later definition within a new scope. A definition within a new scope overrides any previous tag definition appearing in an outer scope. This use of declaring tags is called vacuous structure tag declaration. The declaration does not require the size of the structure as determined by the structure member list. Using such declarations, you can eliminate ambiguity when forward referencing tag identifiers. The following example shows such a case:

```
struct  ambiguous {...};

{
   struct ambiguous;    /*  Vacuous structure tag declaration. */
                        /*  Ignore previous tag currently in scope. */

   struct  inner
   {
      struct ambiguous *pointer;  /* Declare a structure pointer by */
       .                          /* forward referencing.          */
       .
       .
   };

   struct ambiguous     /* Vacuous declaration refers to this  */
   {...};               /* structure, not to the first one declared. */
}
```

  In the example, the pointer to the structure defined using the tag ambiguous points to the second declaration of ambiguous, not to the first.

Structures and unions can contain other structures and unions. For example:

```
struct person
   {
      char first[20];
      char middle[3];
      char last[30];
      struct
      {
          int day;
          int month;
          int year;
      } birth_date;
   }  george, mary;
```

7.12.2 Referencing Members of Structures or Unions

A reference to a member of a structure must be a fully qualified or a pointer-qualified reference. For example, the fully qualified references to the members last and year from the example in the previous section, are as follows:

```
strcpy(george.last, "Harrison");
george.birth_date.year = 1944;
```

A member name denotes the member's data type and its offset from the base of the structure. There are no restrictions on the reuse (as a member name) or redeclaration of a particular name except that the same name cannot be used for more than one member in the same structure.

In VAX C, and other recent compilers, a structure or union reference must be completely qualified; that is, you must prefix a member name in a reference either with a pointer qualifier (pointer-name –>) or with the name of the structure or union and the names of all intervening members. Consider the following structure declaration:

```
main()
{
   struct
      {
         struct { int a1,a2,a3; }  mema;
         struct { int a1,a2,a3; }  memb;
      } *pointer, structure;
   pointer = &structure;

   structure.mema.a1 = 1;          /* Unambiguous                */
   pointer->memb.a1 = 2;

   structure.a1 = 3;               /* Ambiguous: which "a1"?     */
   pointer->a1 = 4;
}
```

Member a1 must be uniquely qualified as being a member of structure mema or structure memb. In fact, structure members that are themselves structures must be given variable identifiers (mema and memb) to make it possible to construct fully qualified references.

A member name is unique if it conforms to either of the following requirements:

- It is used only once.
- If it is used more than once (in different structures), every use denotes a member of the same data type and at the same offset from the base of its structure.

If you use member names that refer to different structures than those in which they were declared, a programming practice that is not recommended, the compiler assumes that the program has an error and issues diagnostic messages. The following checks apply to using member names for reference to structures and unions in which they are not declared:

- If a member name is unique, you can use it in a reference to a structure that it is not a member of, since the address and size of the referenced data can be determined without ambiguity. However, the compiler issues a nonfatal warning message. This usage is maintained for compatibility with other C implementations.
- If a member name is not unique (ambiguous), its use in such a reference causes a fatal error message.

7.12.3 Initializing Structures

In structure declarations, initializers follow the structure variables, not the members. Separate initializing values with commas; delimit them with braces ({}). See Section 7.9 for more information about comma lists.

An example that initializes two structure variables is as follows:

```
struct
   {
      int i;
      float c;
   }  a = { 1, 3.0e10 },  b = { 2, 1.5e5 };
```

The compiler assigns structure initializers in increasing member order. If there are fewer initializers than members for a **static**, **external**, or **globaldef** structure, the structure is padded with zeros. For an **auto** structure, the contents of the uninitialized members are undefined. For more information about storage classes, see Chapter 8.

NOTE

There is no way to specify iterations of an initializer or to initialize a member in the middle of a structure without also initializing the previous members.

Example 7–1 shows these initialization rules applied to an array of structures.

Example 7–1: The Rules for Initializing Structures

```
main()
{
   int l, m;
   static struct
      {
         char ch;
         int i;
         float c;
      } ar[2][3] =
❶        {
❷           {
❸              { 'a', 1, 3e10 },
               { 'b', 2, 4e10 },
               { 'c', 3, 5e10 },
           }
        };

   printf("row/col\t ch\t i\t      c\n");
   printf("-----------------------------------\n");
   for (l = 0; l < 2; l++)
      for (m = 0; m < 3; m++)
         {
            printf("[%d][%d]:", l, m);
            printf("\t %c \t %d \t %e \n",
                   ar[l][m].ch, ar[l][m].i, ar[l][m].c);
         }
}
```

Key to Example 7–1:

❶ You must delimit the initialization of each array row with braces.

❷ You must delimit a structure initialization with braces.

❸ You must delimit an array initialization with braces.

Example 7–1 writes the following output to **stdout**:

```
row/col  ch      i            c
-----------------------------------
[0][0]:  a       1       3.000000e+10
[0][1]:  b       2       4.000000e+10
[0][2]:  c       3       5.000000e+10
[1][0]:          0       0.000000e+00
[1][1]:          0       0.000000e+00
[1][2]:          0       0.000000e+00
```

7.12.4 Variant Structures and Unions

Variant structure and union declarations allow you to reference members of nested aggregates without having to reference intermediate structure or union identifiers. When you nest a variant structure or union declaration within another structure or union declaration, the enclosed variant aggregate ceases to exist as a separate aggregate, and VAX C reproduces its members to the enclosing aggregate.

You declare variant structures and unions using the **variant_struct** and **variant_union** keywords. The format of these declarations is the same as regular structures or unions with the following exceptions:

- You must nest variant aggregates within other valid structure or union declarations.
- You cannot use a tag in a variant aggregate declaration.
- You must provide a variable identifier in the variant aggregate declaration.

Consider the following code example, which does not use variant aggregates:

```
/*  The numbers to the right of the code represent the byte offset  */
/*  from the enclosing structure or union declaration.              */
struct  TAG_1
{
   int   a;           /* 0-byte  enclosing_struct offset */
   char  *b;          /* 4-byte  enclosing_struct offset */
   union  TAG_2       /* 8-byte  enclosing_struct offset */
   {
      int  c;         /* 0-byte  nested_union offset */
      struct  TAG_3   /* 0-byte  nested_union offset */
      {
         int  d;      /* 0-byte  nested_struct offset */
         int  e;      /* 4-byte  nested_struct offset */
      }  nested_struct;
   } nested_union;
} enclosing_struct;
```

If you want to access nested member d, you need to specify all of the intermediate aggregate identifiers, as follows:

```
enclosing_struct.nested_union.nested_struct.d
```

If you try to access member d without specifying the intermediate identifiers, you are accessing the incorrect offset from the incorrect structure. Consider the following example:

```
enclosing_struct.d
```

If you specify this notation, VAX C uses the address of the original structure (enclosing_struct), and adds to it the assigned offset value for member d (0 bytes), even though VAX C calculated the offset value for d according to the nested structure (nested_struct). Consequently, VAX C accesses member a (0 byte offset from enclosing_struct) instead of member d.

The following example shows the same code using variant aggregates:

```
/*  The numbers to the right of the code present the byte offset  *
 *  from enclosing_struct.                                        */
struct  TAG_1
{
   int   a;           /* 0-byte  enclosing_struct offset */
   char  *b;          /* 4-byte  enclosing_struct offset */
   variant_union
   {
      int  c;         /* 8-byte  enclosing_struct offset */
      variant_struct
      {
         int  d;      /* 8-byte   enclosing_struct offset */
         int  e;      /* 12-byte  enclosing_struct offset */
      }  nested_struct;
   }
nested_union;
} enclosing_struct;
```

The members of variant aggregates nested_union and nested_struct are propagated to the immediately enclosing aggregate (enclosing_struct). The variant aggregates cease to exist as individual aggregates.

Since variant aggregates nested_union and nested_struct do not exist as individual aggregates, you cannot use tags in their declarations nor can you use their identifiers (nested_union and nested_struct) in any reference to their members. However, you can reuse the identifier names in other declarations and definitions within your program.

If you need to access member d, use the following notation:

```
enclosing_struct.d
```

If you use the following notation, unpredictable results occur:

```
enclosing_struct.nested_union.nested_struct.d
```

If you use regular structure or union declarations within a variant aggregate declaration, VAX C reproduces the structure or union to the enclosing aggregate, but the members remain a part of the nested aggregate. For instance, if the nested structure in the last example is of type **struct**, the following offsets will be in effect:

```
struct  TAG_1
{
   int   a;           /* 0-byte  enclosing_struct offset */
   char  *b;          /* 4-byte  enclosing_struct offset */
   variant_union
   {
      int  c;         /* 8-byte  enclosing_struct offset */
      struct  TAG_2   /* 8-byte  enclosing-struct offset */
      {
         int  d;      /* 0-byte   nested_struct offset */
         int  e;      /* 4-byte   nested_struct offset */
      }  nested_struct;
   } nested_union;
} enclosing_struct;
```

NOTE

Variant structures and unions are VAX C extensions so they are not portable.

7.12.5 Bit Fields

A structure member can consist of a specified number of bits, called a field, which can be named or unnamed. Use a colon (:) to separate the member's declarator (if any) from a constant expression that gives the field width in bits. No field can be longer than 32 bits (1 longword) in VAX C.

If no field name precedes the field-width expression, it indicates an unnamed field of the specified width. Since nonfield structure members are aligned on byte boundaries, this form can create unnamed gaps in the structure's storage. As a special case, an unnamed field of width 0 causes the next member (generally another field) to be aligned on a byte boundary.

Bit fields must be **unsigned** or **int** data types. The use of other data types is an error. Signed bit fields of the type **int** are recognized by VAX C. There are no restrictions on the use of fields except as follows:

- You cannot declare arrays of fields.
- The ampersand operator (&) cannot be applied to fields, so there cannot be pointers to fields.

Sequences of bit fields are packed as tightly as possible. In VAX C, fields are assigned from right to left.

For example, consider the alignments resulting from the following code:

```
static struct
   {
      char c;
      short int i;
      unsigned fld1 : 3;
      unsigned fld2 : 4;
      unsigned      : 0;
      unsigned fld3 : 4;
   }  a = { 'A', 1024, 06, 012, 014 } ;
```

Member a.i. is aligned on the third byte (at bit 16), because structure elements of the **short** data type are aligned on word boundaries by default. (The preprocessor directive **#pragma member_alignment** can alter this default behavior.) Fields a.fld1 and a.fld2 are packed as tightly as possible in the second longword. The unnamed, 0-length field preceding member a.fld3 causes that field to be aligned on the next byte boundary (bit 40, in the second longword).

7.13 The void Keyword

The **void** keyword is a special data-type specifier that you use in function definitions and declarations for the following purposes:

- To specify a function that does not return a value
- To specify a function prototype with no arguments
- To specify a generic pointer

The following example shows how to use **void** to specify a function that does not return a value:

```
void  message( )
{
   printf("Stop making sense!");
   return;
}
```

The following example shows how to use **void** to specify a function prototype definition that takes no arguments:

```
char function_name( void )
{ return 'a'; }
```

The following example shows a function prototype of a function that accepts the address of a pointer to any object as its first and second arguments:

```
void memcopy (void *dest, void *source, int length);
```

For more information about the **void** data type and function prototypes, refer to Chapter 4.

7.14 The typedef Keyword

Use the **typedef** keyword to define an abbreviated name, or synonym, for a lengthy type definition. In such a declaration, the identifiers name types instead of variables. For example:

```
typedef char CH, *CP, STRING[10], CF();
```

In the scope of this declaration, CH is a synonym for character, CP is a pointer to a character, STRING is a 10-element array of characters, and CF is a function returning a character. You can use each of the type definitions in that scope to declare variables, as follows:

```
CF      c;              /* "c": Function returning a character  */
STRING  s;              /* "s": 10-character string             */
```

7.15 Interpreting Declarations

The VAX C programming language syntax for declaring objects is unlike the declaration syntax of other languages. Since the exact meaning of a complicated VAX C declaration is not always apparent, even to an experienced C programmer, this section gives guidelines for interpreting and constructing VAX C declarations.

VAX C uses the same set of operators and symbols for declarators as for identifiers in an expression. The following example declares integer x and pointer px:

```
int  x;
int  *px;
```

Declarator *px has the same form as that used to yield an integer in an expression. Consider the following example:

```
x  =  *px;
```

In the case of simple declarators, this symmetry makes it easy to determine the type of an expression or the meaning of a declarator. Expression *px results in the integer object that px points to.

Complicated declarators can be difficult to interpret without some additional guidelines. The important one to remember is that the symbols used in declarators are C operators, subject to the usual rules of precedence and grouping (associative nature). In order of precedence, the operators used in declarators are as follows:

- The primary-expression operators (( )) for "function returning . . . " and ([ ]) for "array of . . . ", where the ellipsis indicates the type specified in the declaration.

  These operators group from left to right.

- The unary asterisk (*), for indirection or "pointer to . . . ", which groups from right to left.

Consider the following example:

```
int  *x[];
```

Even this brief declaration may be confusing. Does it declare an array of pointers to integers, or a pointer to an array of integers? Since the brackets are of higher precedence, it follows that:

- *x[ ] is an integer
- x[ ] is a pointer to an integer
- x is an array of pointers to integers

You can interpret most complicated declarators and expressions quickly by using such a sequential breakdown. Note that the asterisk was removed before the brackets because it is of lower precedence.

Also note that this interpretation process enumerates all the possible usage constructs involving a declarator and giving the semantic interpretation.

When constructing or interpreting declarations or expressions, use the following scheme[1] for translating the meanings of the operators:

- "*" == "pointer to"
- "( )" == "function returning"
- "[ ]" == "array of"

Consider the following example:

```
char  *x()[];
```

The breakdown is as follows:

- *x( )[ ] is of type **char**
- x( )[ ] is (pointer to) **char**
- x( ) is (array of) (pointer to) **char**
- x is (function returning) (array of) (pointer to) **char**

In the third step, the bracket operator is removed first because primary-expression operators are of equal precedence and group from left to right. That is, "( )[ ]" means "function returning array of", not "array of function returning . . . ".

[1] Bruce Anderson, "Type Syntax in the Language C: An Object Lesson in Syntactic Innovation," *SIGPLAN Notices* 15, No. 2 (March 1980).

As a general rule, when breaking down a declaration in this manner, remove the operators with the lowest precedence first. Then, if the operators are of equal precedence and group from left to right, remove the rightmost operator first; if they group from right to left, remove the leftmost operator first.

The declaration shown is semantically invalid; VAX C allows functions returning addresses of arrays, but not functions returning arrays. Perhaps the programmer intended to specify a function returning the address of an array of pointers to characters. To make the declaration valid, start at the bottom of a breakdown and work back to a valid declaration as follows:

- x is (function returning) (pointer to) (array of) (pointer to) **char**
- x( ) is (pointer to) (array of) (pointer to) **char**
- *x( ) is (array of) (pointer to) **char**
- (*x( ))[ ] is (pointer to) **char**
- *(*x( ))[ ] is **char**
- **char** *(*x( ))[ ];

In the final declaration, the first asterisk (since it groups right to left) applies to **char**.

Use parentheses in declarations along with the function parameter-list operator (( )) to change the binding of operators. For example, the outer parentheses introduced in the fourth step of the previous example prevent the brackets from binding to the inner set of parentheses.

Consider the following example:

```
char  (* (*x()) []) ();
```

The breakdown is as follows:

- (* (*x( )) [ ]) ( ) is **char**
- * (*x( )) [ ] is (function returning) **char**
- (*x( )) [ ] is (pointer to) (function returning) **char**
- *x( ) is (array of) (pointer to) (function returning) **char**
- x( ) is (pointer to) (array of) (pointer to) (function returning) **char**
- The identifier x is a function returning a pointer to an array of pointers to functions returning characters

Spaces are used in this example to separate the declarator into its component parts. Since spaces, tabs, and newlines are ignored by the parser, use them in declarations for clarity.

Tables 7–5 and 7–6 provide examples of legal and illegal VAX C declarations.

Table 7–5: Legal C Declarations

Declaration	Meaning
int i;	An **int**
int *p;	Pointer to **int**

(continued on next page)

Table 7–5 (Cont.): Legal C Declarations

Declaration	Meaning
int a[ ];	Array of **int**
int f( );	Function returning **int**
int **pp;	Pointer to pointer to **int**
int (*pa)[ ];	Pointer to an array of **int**
int (*pf)( );	Pointer to a function returning **int**
int *ap[ ];	Array of pointer to **int**
int aa[ ][ ];	Array of an array of **int**
int *fp( );	Function returning pointer to **int**
int ***ppp;	Pointer to a pointer to a pointer to **int**
int (**ppa)[ ];	Pointer to a pointer to an array of **int**
int (**ppf)( );	Pointer to a pointer to a function returning **int**
int *(*pap)[ ];	Pointer to an array of pointer to **int**
int (*paa)[ ][ ];	Pointer to an array of an array of **int**
int *(*pfp)( );	Pointer to a function returning pointer to **int**
int **app[ ];	Array of pointer to pointer to **int**
int (*apa[ ])[ ];	Array of pointer to an array of **int**
int (*apf[ ])( );	Array of pointer to a function returning **int**
*aap[ ][ ];	Array of an array of pointer to **int**
int aaa[ ][ ][ ];	Array of an array of an array of **int**
int ***fpp( );	Function returning a pointer to pointer to **int**
int (*fpa( ))[ ];	Function returning a pointer to an array of **int**
int (*fpf( ))( );	Function returning a pointer to a function returning **int**

Table 7–6: Illegal Declarations

Declaration	Meaning
int af[ ]( );	Array of function returning **int**
int *fa( )[ ];	Function returning an array of **int**
int ff( )( );	Function returning a function returning **int**
int (*paf)[ ]( );	Pointer to an array of a function returning **int**
int (*pfa)( )[ ];	Pointer to a function returning an array of **int**
int (*pff)( )( );	Pointer to a function returning a function returning **int**
int aaf[ ][ ]( );	Array of an array of a function returning **int**
int *afp[ ]( );	Array of a function returning a pointer to **int**
int afa[ ]( )[ ];	Array of a function returning an array of **int**
int aff[ ]( )( );	Array of a function returning a function returning **int**
int *fap( )[ ];	Function returning an array of pointer to **int**
int faa( )[ ][ ];	Function returning an array of an array of **int**

(continued on next page)

Table 7–6 (Cont.): Illegal Declarations

Declaration	Meaning
int faf( )[ ]( );	Function returning an array of a function returning **int**
int *ffp( )( );	Function returning a function returning pointer to **int**
int *ffa( )( )[ ];	Function returning a function returning pointer to an array of **int**
int fff( )( )( );	Function returning a function returning a function returning **int**